2017

Optimizing Quadratic Functions over the Set of Permutations

Yutong Chang
Lehigh University

Optimizing Quadratic Functions over the Set of Permutations

by

Yutong Chang

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Departement of Industrial and Systems Engineering

Lehigh University

May 2017

Thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science in Management Science and Engineering.

Thesis Title: Optimizing Quadratic Functions over the Set of Permutations

Student Name: Yutong Chang

_____

Date Approved

_____

Martin Takáč (Thesis Advisor)

_____

Tamás Terlaky (Department Chair Person)

Acknowledgements

I would like to thank my thesis advisor Dr. Martin Takáč, professor of our department, Department of Industrial Systems and Engineering at Lehigh University. During the school year when I was doing my research project for thesis, Prof. Takáč has been always there whenever I ran into a trouble spot or had a question about my research. He consistently allowed this paper to be my own work, but guided me to the right direction whenever he thought I needed it.

I would also like to thank Dr. Tamás Terlaky, Chair of our department and other professors in our department. They give me very helpful advice during the department symposium and help me understand my research problem better.

Also I want to thank Cong Han Lim, the author of [17], who provided me the code used in their paper for comparing the performance of different method and answered my questions highly patiently.

And finally I must express my very deep gratitude to my parents and to my friends for providing me with unfailing support and continuous encouragement throughout my years of study, my study life abroad, and through the process of researching and writing the thesis. My accomplishment of master study may not have been possible without them.

Thank you!

Contents

List of Tables

List of Figures

Abstract

Seriation problem is widely used in many fields like archeology[20] and shotgun gene sequencing[9, 19]. It aims to reorder a linear permutation based on given similarity information and it is an optimization problem over the set of permutation. Due to the large size of feasible set and the variable type, the seriation is an NP-hard quadratic mixed integer programming(MIP) problem. In order to solve this problem efficiently, a construction proposed recently by Goemans[11], sorting network is used to constrain the solutions of the problem to be permutation and reformulate the problem. And we solve the MIP problem using heuristic method and branch and bound method and compare their performance.

Chapter 1

Introduction

1.1 Background

In seriation problem, a similarity matrix is given for a set of variables in a permutation where the similarity between variables decreases with their distance in the permutation, and the problem seeks to sort the linear permutation based on the given similarity information. As is shown in Figure 1.1, the left figure is a random permuted permutation with a given similarity information, we can gain nothing from this figure. While the right one is the permutation reordered by the given similarity information where we can obtain some information from it.

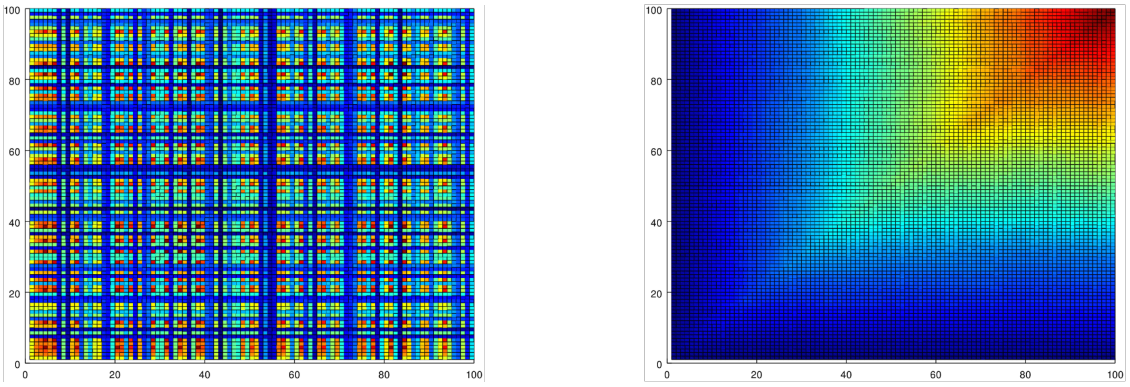

Figure 1.1: Plot of information matrix of permutation of length 100. The left one is the matrix with rows and columns randomly permuted and the right one is the rearranged matrix.

When solving optimization problems over the set of permutation, Birkhoff polytope, the convex hull of the set of permutation matrices, is often used to formulate relaxation of the original problem. One of the example of formulating relaxations using Birkhoff polytope is in [7]. $O(n^2)$ variables are required to representing the Birkhoff polytope, which is much more than $O(n)$ variables needed in vector form of permutation. Fogel[8] discussed the convex relaxation based on Birkhoff polytope form in their paper on seriation problem.

1.2 Seriation Problem

In general, seriation refers to data analysis technique that reorder objects in a linear sequence to reveal the most regularity and pattern among the system[16]. Suppose we have n objects ordering linearly, with a similarity function where the function value increases with the distance between elements in the line. The similarity matrix is a n-squre matrix where the (i, j) element is the measure of similarity between the ith and jth element within the line. Given a symmetric similarity matrix, the seriation problem can be expressed in the following way.

$$\min_{\pi \in P^n} \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij}(\pi(i) - \pi(j))^2, \tag{1.1}$$

where P^n denotes the set of all permutation vectors of length n. We define Laplacian matrix to be $L_A = diag(A\mathbf{1} - A)$, so we can write the objective function 1.1 into following matrix form

$$\min_{\pi \in P^n} \pi^T L_A \pi \tag{1.2}$$

where $\pi \in P^n$ denotes a general permutation whose components is $\pi(i)$, $i = 1, 2, ..., n$, and $\mathbf{1}$ denotes the n-element vector whose components are all ones.

The seriation problem has been proved to be NP-complete in [10] on the algorithm front.

1.3 Sorting Network

We construct sorting network[6] to confine the sequences in the feasible set to be permutation. Sorting network are comparison network that can sort its input. Comparison network consists of wires and comparators. Numbers are transmitted through wires and then making comparison in comparators. As shown in Figure 1.2, a comparator swap values if and only if the value on the top wire is smaller than the one on the bottom.

Figure 1.2: A comparator with inputs x, y and outputs x' and y'

In other word, it performs the function below:

$$x' = \min(x, y)$$
$$y' = \max(x, y)$$

We assume a comparason network has n input sequence $[a_1, a_2, ..., a_n]$ and a corresponding output sequence $[b_1, b_2, ...b_n]$ and they transmitted through n wires. And each comparator takes $O(1)$ time to operate comparison. Now we define depth within the network and the depth of the whole network. An input wire has depth of 0 and depth increase 1 after going though a comparator. So if a comparator has two input wires with depth d_1 and d_2, then the depth of this comparator is the depth of its output wire of this comparator, which is $max(d_1, d_2) + 1$. Hence the depth of a comparison network is the maximum depth of its output wires. Figure 1.3 shows an example of how a simple 4-input comparison network works and how the depth change inside it.

A sorting network is a comparison network where the output of the network is monotonically increasing. We first introduce the zero-one principle to make illustration of how these comparison network works more clearly. Then bitonic sorting network, merging sorting network are introduced to build the final sorting network.

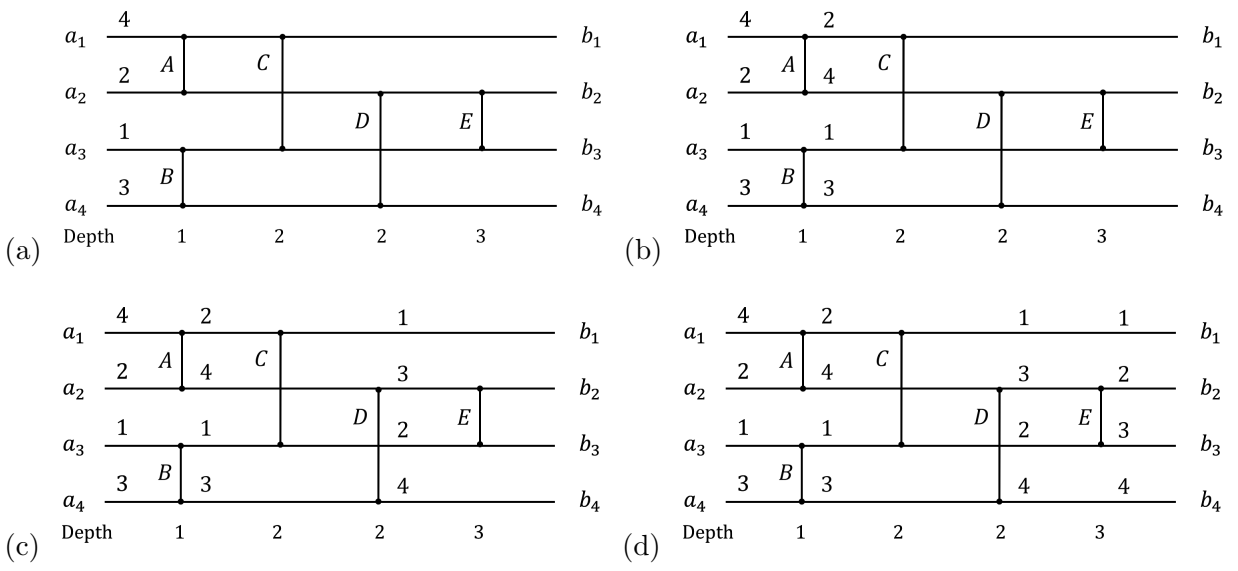

Figure 1.3: (a) A 4-inputs comparason network with input sequence $[4, 2, 1, 3]$, 5 comparators and 3 depth. (b) Input sequence made comparison after going though the first-depth-comparator A and B and got sequence$[2, 4, 1, 3]$ at time 1. (c) The sequence made comparison at second-depth-comparator C and D at time 2, getting following sequence $[1, 3, 2, 4]$. (d) The sequence made final comparison at time 4 after passing comparitor E, the last depth comparator, getting the sorted output sequence $[1, 2, 3, 4]$.

1.3.1 The zero-one principle

The zero-one principle indicates that if a sorting network can sort correctly for inputs with elements solely from the set of $\{0, 1\}$, it can correctly sort any input with arbitrary number. So zero-one principle allows us to construct sorting network only take into account inputs composed of 0's and 1's. That is, if a n-input sorting network can sort all 2^n possible sequences of 0's and 1's, it can work correctly for any input.

There are numerous choices of sorting network. The most compact one is the AKS sorting network[1] with $O(nlogn)$ comparators but it is impractical because of the difficulty to construct it. Instead, we would like to choose network with slightly worse complexity. Batcher[4] introduced the bitonic sorting network with $O(n \log^2 n)$ which is more practical. The construction of constraints describing the sorting network can be simply recursively performed that runs in $O(n \log^2 n)$ time. So we choose the bitonic sorting network here and got fairly good performance in the implementations.

1.3.2 Bitonic sorting network

A bitonic sorting network is a network which can sort any bitonic sequence. A bitonic sequence[21] is a sequence where $a_1 \leq a_2 \leq ... \leq a_k \geq a_{k+1} \geq ... \geq a_n$, $0 \leq k < n$, or can be circularly shifted to be such a sequence. And any sequence of just one or two numbers is also a bitonic sequence. A bitonic sorter is composed of half-cleaners.

The half-cleaner

Half-cleaners are n-input comparison network where input in line i compares with input in line i+1/2. So the depth of half-cleaners is 1. for $i = 1, 2, ..., n/2$ and we assume n is even. So when a 0-1 bitonic sequence goes thought a half-cleaner, smaller values go in to top of the the sorted output sequence and the larger ones go on the bottom. And it is proved in [6] that both the top and bottom halves are bitonic and every element of the top half is at least as mall as every element in the bottom half and at least one half is clean - consisting of either all 0's or all 1's. Figure 1.4 gives three examples of 8-input half-cleaner.

The bitonic sorter

The bitonic sorter is built by recursively combining half-cleaners, as shown in Figure 1.5. And Figure 1.6 gives an simple illustration of how a BITONIC-SORTER[8] works after unraveling the

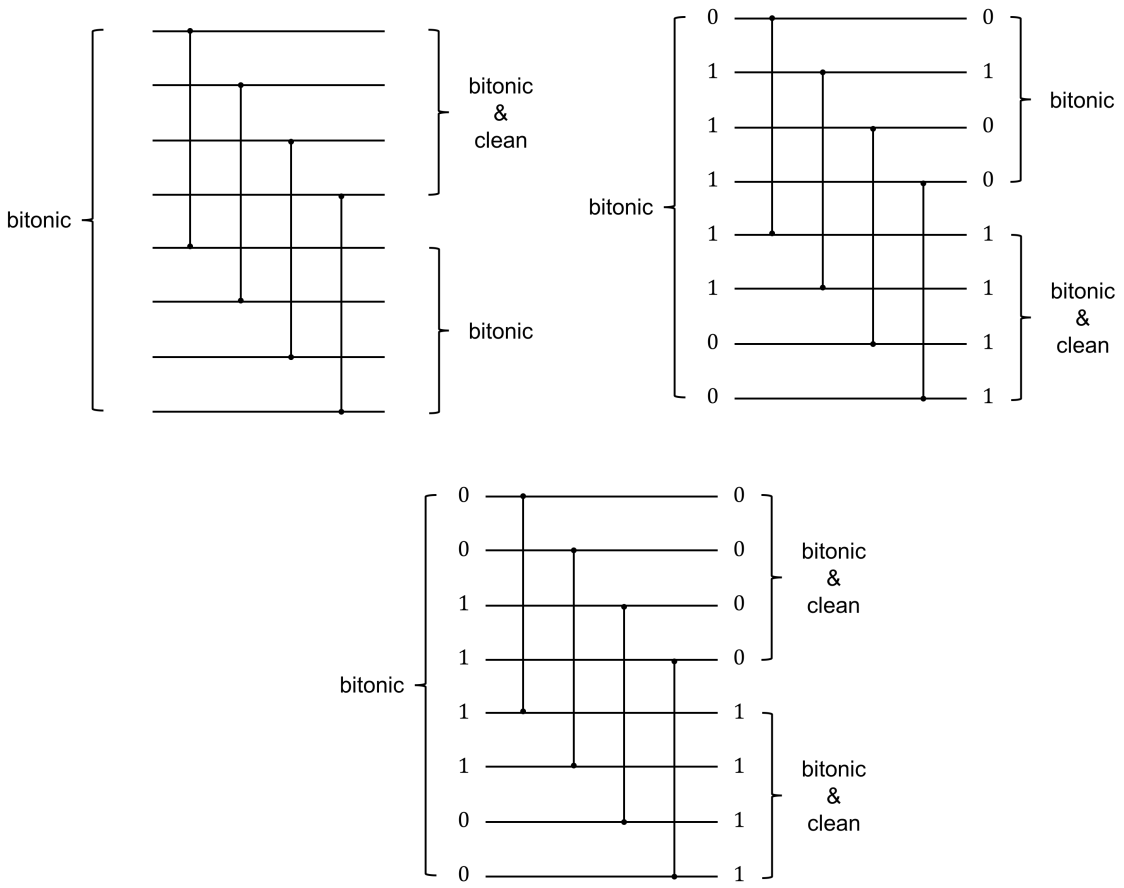

Figure 1.4: Three instances of HALF-CLEANER[N], the 8-input half-cleaner with difference inputs. Both top and bottom halves of output are bitonic and at least one half is clean.

recursion.

The depth $D(n)$ of n-input bitonic-sorter is $D(n) = lg(n)$, given by the recurrence

$$D(n) = \begin{cases} 0 & \text{if } n = 1, \\ D(n/2) + 1 & \text{if } n = 2^k, k \geq 1, \end{cases}$$

So any 0-1 bitonic sequence can be sorted by the bitonic sorter. According to the zero-one principle, the bitonic sorter can sort arbitrary bitonic sequence of arbitrary number.

1.3.3 The merging network

Merging network can merge two sorted sequences into one sorted output sequence. We use merging network to construct the sorting network. We construct the n-input merging network MERGER[n]

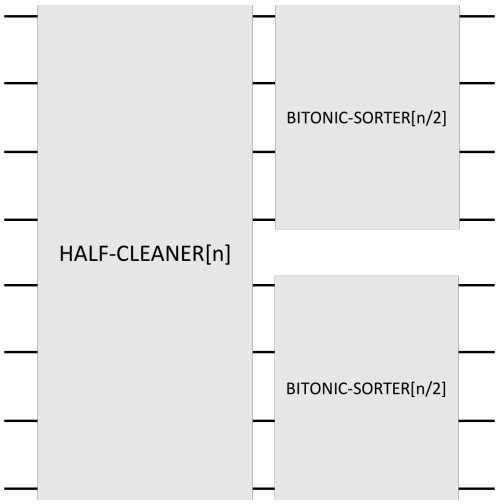

Figure 1.5: Recursive construction of bitonic-sorter.

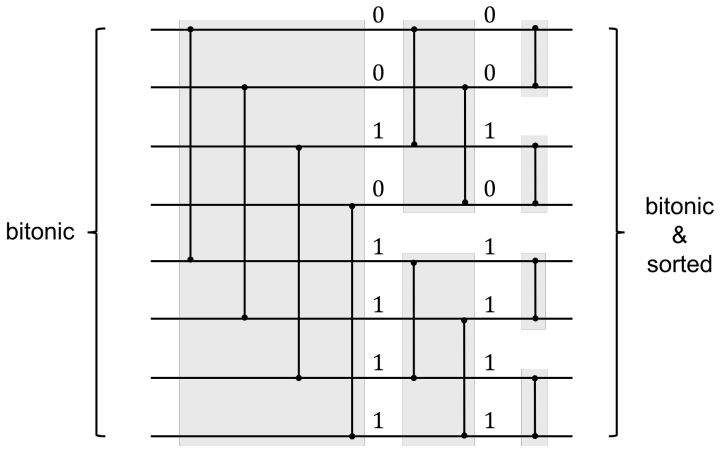

Figure 1.6: BITONIC-SORTER[8] with 0-1 input shows how the recursion works.

based on the BITONIC-SORTER[n]. We can construct the merging network by modifying the first half cleaner of the n-input bitonic sorter. The aim is to reverse the second half of the input to make the input bitonic. So with two sorted sequence $[a_1, a_2, ..., a_{n/2}]$ and $[b_1, b_2, ..., b_{n/2}]$, we would like to sort the bitonic sequence $[a_1, a_2, ..., a_{n/2}, b_{n/2}, ..., b_2, b_1]$. So instead of comparing the inputs i and $n/2 + 1$ for the first half-cleaner, we need to compare the inputs i and $n - i + 1$, as is shown in Figure 1.7.

As shown in Figure 1.9, the final merging network only differ from the the bitonic sorting shown in Figure 1.5 and 1.6 in the first stage. Therefore the depth of a n-input merging network is as same as that of a n-input bitonic sorter, which is $\lg(n)$.

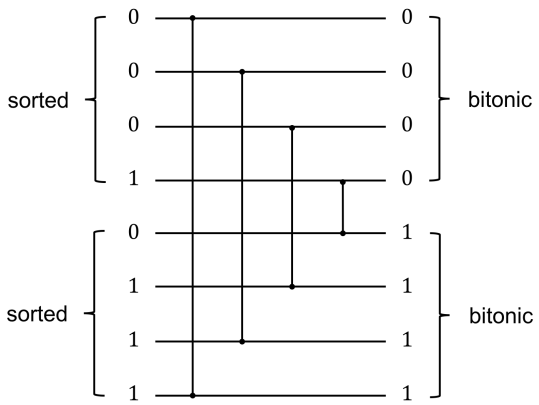

Figure 1.7: The first stage of 8-input merger. It transforms two sorted input to two bitonic ones.

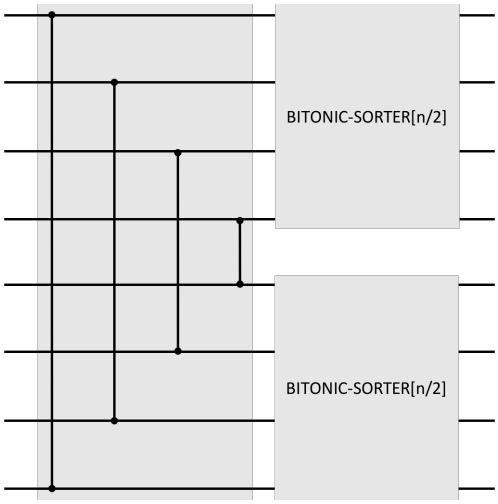

Figure 1.8: Like the recursive construction of bitonic-sorter, the only difference of the construction of merging network is the first stage.

1.3.4 The sorting network

After having all the necessary networks, we now can built the final sorting network we need that can sort arbitrary input sequence. As is shown in 1.10, the n-input sorting network SORTER[n] is built recursively. The inputs are sorted parallelly by two SORTER[n/2] to sort two sub-sequences. Then the two sorted sub-sequences are merged by MERGER[n]. Figure 1.11 shows an example of how a 8-input sorting network is built under unrolled recursion and how it works with 0-1 input

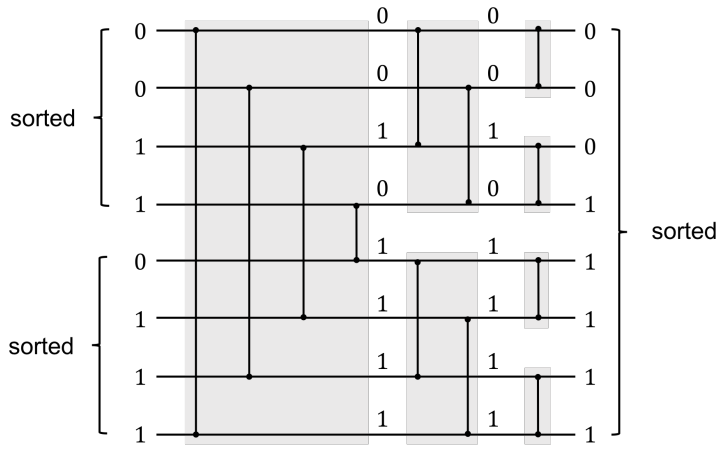

Figure 1.9: The same merging network after uncover the recursion of Figure 1.8

sequence.

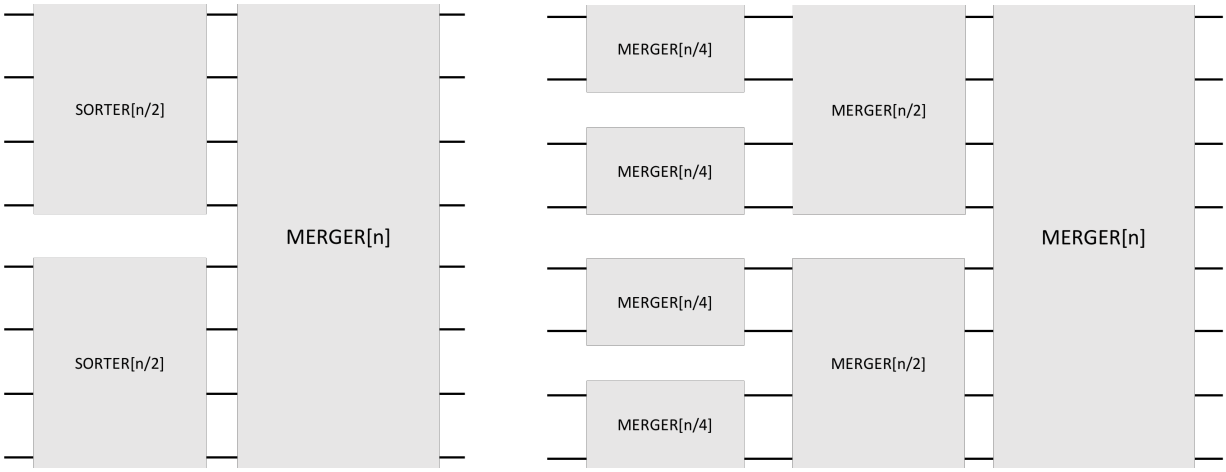

Figure 1.10: The recursive construction of sorting network combining merging networks.

With illustration of Figure 1.11, we can see that the input data go through in total $\lg(n)$ stages in the n-input sorting network. Since each input can be thought as a sorted sequence, we use $n/2$ MERGER[2] to produce sorted sequences of length 2 in the first stage. For the second stage, there are $n/4$ MERGER[4] to merge the sorted sequences from stage 1 to produce 4-element sequences. The same procedure happens in stage 3. Generally for $k = 1, 2, ..., \lg(n)$, stage k contains $n/2^k$ MERGER[2^k] that can merge sorted sequences of length 2^{k-2} to be 2^k-elements sequence. This sorting network can sort any 0-1 sequences, consequently it can sort arbitrary sequences according

to the zero-one principle.

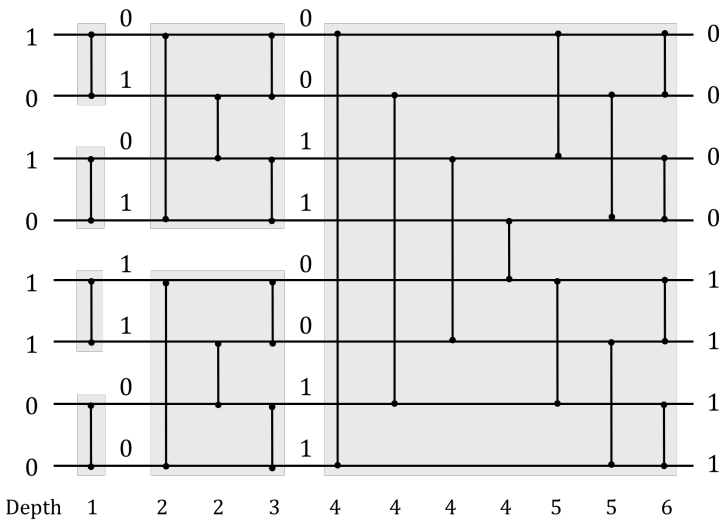

Figure 1.11: The same merging network after uncover the recursion of Figure 1.8. The depth of comparators in each stage is given and the sample 0-1 sequences after each stage is indicated.

The depth of the sorting network can be analyzed recursively. The depth of n-input sorting network is the sum of the depth $D(n/2)$ of $n/2$-input sorting network and the depth of MERGER[n], which is $\lg(n)$. Hence the depth of an n-input sorting network is given by the following recurrence

$$D(n) = \begin{cases} 0 & \text{if } n = 1, \\ \mathrm{D(n/2)+lg(n)} & \text{if } n = 2^k, k \geq 1, \end{cases}$$

Of which the solution is $D(n) = O(\lg^2 n)$. The comparators within one depth works in parallel. Accordingly, we can sort an n-element sequence in $O(\lg^2 n)$ time.

1.3.5 Constraints formulation

As we refered before, the function of a comparator with input x_1, x_2 and output y_1, y_2 can be expressed as follows

$$y_1 = \min(x_1, x_2),$$
$$y_2 = \max(x_1, x_2).$$

That is

$$\begin{cases} y_1 \leq \min(x_1, x_2) \\ y_1 \geq \min(x_1, x_2) \\ y_2 \leq \max(x_1, x_2) \\ y_2 \geq \max(x_1, x_2), \end{cases}$$

which is equivalent to the following formulation

$$\begin{cases} y_1 \leq x_1 \\ y_1 \leq x_2 \\ y_1 \geq x_1 - M * C \\ y_1 \geq x_2 - M * (1 - C) \\ y_2 = x_1 + x_2 - y_1, \end{cases} \tag{1.3}$$

where M is a significantly large number and C is a binary variable.

1.4 Spectral Method

Atkins et al.[3] come up with the idea that finding the second smallest eigenvalue, which is also known as Fiedler value, can also solve the seriation problem.

1.5 The MIP Problem

So the problem we are solving can be formulated as following

$$\min_{\pi \in P^n} \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij}(\pi(i) - \pi(j))^2,$$

or in matrix form

$$\min_{\pi \in P^n} \pi^T L_A \pi,$$

where $\pi \in P^n$ denotes a general permutation whose components is $\pi(i)$, $i = 1, 2, ..., n$, and $\mathbf{1}$ denotes the n-element vector whose components are all ones. As described in last section, in order to confine the π to be permutation, the output π^{out} after π going though the sorting network should satisfy the following constraints

$$\pi_i^{out} = i, for\ i = 1, 2, ..., n. \tag{1.4}$$

1.4 is composed of all the comparators with each comparator $k = 1, 2, ..., n$ written to be in form of 1.3, which is

$$\begin{cases} x^k_{(out,top)} \leq x^k_{(in,top)} \\ x^k_{(out,top)} \leq x^k_{(in,bottom)} \\ x^k_{(out,top)} \geq x^k_{(in,top)} - M * C \\ x^k_{(out,top)} \geq x^k_{(in,bottom)} - M * (1 - C) \\ x^k_{(out,bottom)} = x^k_{(in,top)} + x^k_{(in,bottom)} - x^k_{(out,top)}, \end{cases}$$

where M is a significantly large number and C is a binary variable.

Since we have each variables in the permutation to be integer and the binary variables C for each comparator, the original problem is mixed integer programming(MIP) problem according to the definition[23], which is descrete and non-convex. MIP problem can be hard to solve since the memory needed and solve time can increase exponentially with the size of the problem increasing. And also for a permutation of length n, there exits $n!$ possible arrangement of permutation for

the feasible set of this problem, this will be a fairly large number when n becomes large. A classic and most frequently used method to solve this kind of MIP problem is called branch and bound.

1.5.1 Branch and bound

Branch and bound[22] was first raised by A. H. Land and A. G. Doig[15] for solving discrete programming problem, and has been the most generally used for solving NP-hard optimization problems.It is an algorithm that is made up of systematic enumeration of candidate solutions by relaxing the original problem without some integer or binary constraints, and the set of candidate solution is thought to form a rooted tree with the full set at the root. The algorithm explores the subsets of the whole solution sets which is the branches of the solution tree. For a current solution that has been found, if the decision variable with integer constraints has integer value, then the search stops. If not, the algorithm chooses one variable without integer value and continue "branching" to two sub-problems with more tightly constrainted variable values. And the algorithm checked the estimated upper and lower bound of the branch on the optimal solution, and will discard the branch if a better solution is not found within it than the best solution found so far. The procedure is repeated until the algorithm finds the best solution that satisfies all of the integer constraints is found.

Chapter 2

Experiments

The experiments were run in two different ways. We first solved the problem heuristically in MATLAB R2014b. Then we solve the MIP problem in AMPL with CLPEX on COR@L Lab[5] cluster Polyps. We compare the cpu time and objective function value of both methods and make some change of the AMPL code to improve its performance.

2.1 MATLAB Experiment

In MATLAB we run the heuristic method[18] to solve the MIP problem to get a fairly optimal solution. The main idea here of heuristic method is to start with a random permutation of length n and calculate its objective function value. Then we randomly switch some elements in the permutation and compare its objective function value to the previous one. We repeat the procedure for a certain amount of iteration to get a relatively small objective value(since we are doing minimization). Following is a pseudo code that illustrate how the heuristic algorithm works.

Algorithm 1 Heuristic Method

```
π ← random permutation
Calculate objective function value f(π)
while iteration < max iteration do
  Switch some elements of permutation π to get new π'
  Calculate objective function value f(π')
  if f(π') < f(π) then
    Store f(π')
  end if
end while
```

We also run spectral method described by Atkins et al.[3] using the MATLAB eigs function. The matlab code of spectral method is provided by the authors of [17].

2.2 AMPL Experiment

We use CPLEX[13] solver in AMPL[2] to solve the Quadratic MIP problem.
We first generate data files and model files for AMPL with python. Then we use CPLEX solver in AMPL to solve the problem. Appendix A gives an example of how the AMPL model looks like for matrix data size of 8. Despite of comparing the AMPL performance to MATLAB performance, we also improved the formulation of AMPL model to see their performances.
First we formulate the model as described in Section 1.5. Then we add constraints for each depth of the sorting Network to confine the sum of variable C within a depth to be the total number of comparators within the same depth. Though it's apparent for us to see the relationship between these to should be the same, but it may influence the solve time of CPLEX solver. Then we set initial point of permutation for CPLEX solver. The CPLEX solver use branch and bound to solve the MIP problem hence the initial point for the algorithm will influence the total iteration of branch and bound, which will effect the solve time of the solver.

2.3 Experiment Dataset

2.3.1 Random generated dataset

We use MATLAB to generate the Laplacian of similarity matrix, which is positive semi-definite matrix. We generate matrices of size 2^n, where $n = 4, 5, 6, 7, 8$. For each size, we fixed the random seed in MATLAB from 1 to 100 to generate 100 different matrices. Algorithm 2 is the MATLAB code to generate the dataset.

2.3.2 Munsingen dateset

We also test out solution quality on a specific seriation problem drawn from archaeology[20]. The Munsingen dataset is introduced by Hodson[12], and is manually rearranged in [14]. It consists of a 70×59 binary matrix M indicating the presence of 70 artifact types in 59 graves, where the

Algorithm 2 Generation of random dataset

```
for i=4:9
    n=2^i;
    dirname=int2str(n);
    mkdir(dirname);
    cd(dirname)
    for seed=1:100
        rng(seed);
        G = rand(n,2*n);
        A = G*G';
        D=diag(A);
        D=D.^(-0.5);
        D=diag(D);
        A=D*A*D;
        filename=strcat(int2str(n), '_', int2str(seed), '.dat');
        save(filename,'A','-ASCII');
    end
end
```

artifacts is assumed to be associated with some particular time period. The goal of solving the seriation problem on this dataset is to reorder the graves by time. We minimize the objective function over MM^T. Following Figure 2.1 is the plot of the sorted Munsingen data matrix M(left) and the matrix with rows random permuted.

Figure 2.1: The plot of Munsingen data matrix M.(From [17]) The left figure shows the Munsingen data matrix M while the right plot shows the matrix with rows randomly permuted.

2.4 Experiment Results

Comparison with Random Dataset

We first compare the CPU time and objective function value of MATLAB and CPLEX solver, the results are shown relatively in table 2.1 and table 2.2. And we also computed the relative improvement to MATLAB results.

We can see that the CPLEX solver can find the optimal solution of the original MIP problem while MATLAB can only get a relatively optimal solution. And the CPU time of CPLEX solving the MIP problem is much more faster that of MATLAB, as is shown in 2.2.

Table 2.1: Comparason of CPU time to solve the MIP problem in MATLAB and AMPL, where for AMPL model we only have constraints for comparators.

CPU time			
N	**matlab**	**ampl**	**Relative Improvement**
4	1.14	0.03	0.97
8	3.44	0.01	1.00
16	3.57	0.02	0.99
32	3.58	0.18	0.95
64	9.24	3.98	0.57
128	13.92	2.45	0.82
256	50.02	18.27	0.63
512	280.41	9.05	0.97

Table 2.2: Comparason of objective function values of the MIP problem in matlab and AMPL, where for AMPL model we only have constraints for comparators.

Objective function value			
N	**matlab**	**ampl**	**Relative Improvement**
4	80.91	80.37	0.67%
8	1057.64	1033.27	2.30%
16	14338.35	13705.08	4.42%
32	209023.94	204419.94	2.20%
64	3259609.88	3198072.41	1.89%
128	50990143.74	50495095.50	0.97%
256	813277709.55	806220627.90	0.87%
512	12956217881.44	12870856970.00	0.66%

Then we add to the AMPL model the constraints which confine the sum of C within each depth to be the total number of comparators within the depth. Since adding constraints will not

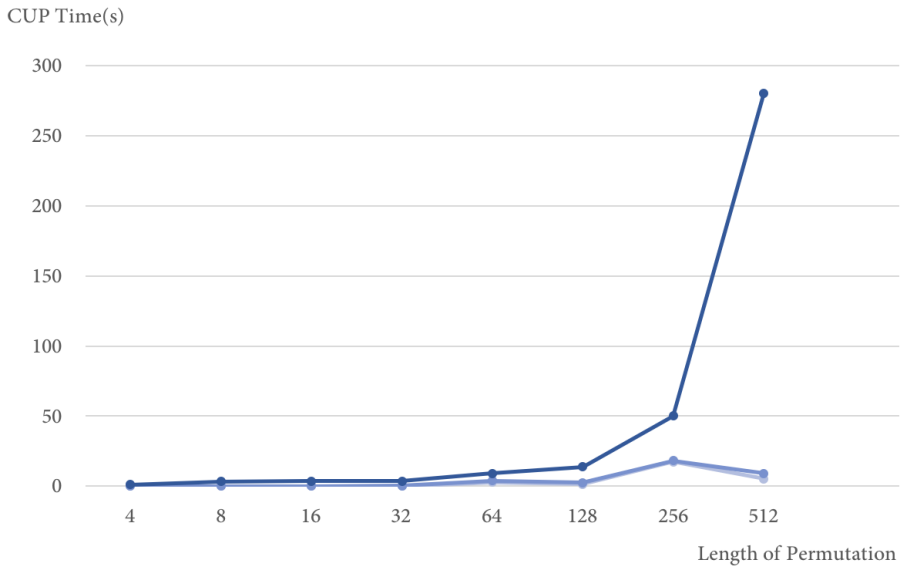

Figure 2.2: CPU time of MATLAB and CPLEX solver. The upper line is the CPU time of MATLAB, which increase fast with the increment of problem size. The line on the bottom is the CPU time of CPLEX solver, which does not have apparent increment with the size of problem.

effects the objective function value, we only compare the CPU time of the two AMPL models. Table 2.3 shows the results. We can see that though the constraints we add is obvious and seems to be common sense to us, it helps the CPLEX solver to solve the model faster than before.

Table 2.3: Comparison of CPU times of the two AMPL models.

	CPU time	
N	**Original**	**Add Constraints**
4	0.032	0.024
8	0.012	0.008
16	0.020	0.016
32	0.184	0.108
64	3.976	3.192
128	2.448	1.508
256	18.272	16.660
512	9.052	4.372

Then we use the solution we found using heuristic method in MATLAB as the initial point for CPLEX solver to see how it effects the iteration number of branch and bound. From Table 2.4 we can found for large size of problem, setting the initial point will help to reduce the number of branch and bound iteration, which will lead to the decrease of solve CPU time.

Table 2.4: Comparison of branch and bound iterations of the two AMPL models.

Number of iterations of B&B		
N	**Without initials**	**With initials**
4	37	37
8	0	0
16	0	0
32	0	0
64	5477	4431
128	0	0
256	5949	5293
512	0	0

Then we compare the results of spectral method and the reformulated model using sorting network solved by CPLEX solver. We can found that the objective function value got from the sorting netwrok formulation solved by CPLEX is always better than the results solved by spectral method in MATLAB. Figure 2.3 shows the boxplot of the percentage improvement.

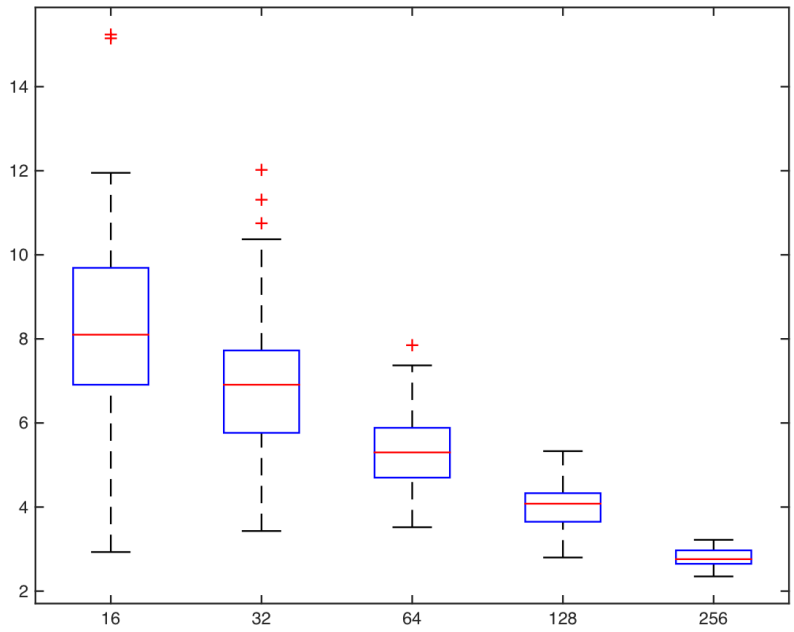

Figure 2.3: The box plot of percentage improvement of sorting network formulation solved by CPLEX compared to the formulation solved by spectral method.

2.5 Additional Empirical Observation

For the random generated dataset, the matrices are positive semi-definite with 1's at diagonal and values between 0 and 1 for non diagnal elements. If we use these matrices A generated directly as the Laplacian matrix, which is for

$$\min_{\pi \in P^n} \pi^T A \pi.$$

The CPLEX can find the optimal solution within seconds for size of hundreds. But when we use A as the original matrix in

$$\min_{\pi \in P^n} \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij}(\pi(i) - \pi(j))^2$$

and get the Laplacian matrix by using $L_A = diag(A\mathbf{1} - A)$, where in L_A the diagonal elements are positive and the rest is negitive number between -1 and 0, the CPLEX solver will take forever to find the optimal solution. The CPLEX solver will stop branch and bound when the relative MIP gap between the objective function value and the best bound reaches a small number. For the later case, even when the size of the problem is small, for example $n = 16$, the best bound CPLEX found increase extremely slowly from zero and the objective decrease very slowly from a very large number. So the relative MIP gap will never reach the small number needed to solve the optimization problem. This problem may be caused by the negative number inside the Laplacian matrix. Further work needed to be done to find out the reason that may lead to this slow down for CPLEX solver. Although the CPLEX solved the problem with calculated Laplacian matrix very slowly, the objective function value it get within a set time limit is better than spectral method. Table 2.5 shows the compared objective function value of sorting network formulation and spectral.

Table 2.5: Objective function value of sorting network formulation solved by CPLEX and spectral method.

n	Spectral	SN	Reletive Error
4	30.64	28.89	5.70%
8	499.03	445.08	10.81%
16	8169.20	7677.61	6.02%
32	131642.87	127635.72	3.04%
64	2099429.70	2083363.49	0.77%
128	33542813.54	33435000.00	0.32%

Chapter 3

Conclusions and Future Work

In this thesis, Goemans' compact description of permutation using sorting network into quatratic optimization. Compared to the commonly used method - heuristic method and other method that has been used like spectral method, reformulating the MIP problem using sorting Network can solve the NP-hard quadratic MIP problem over the set of permutation more efficiently. Also the

Bibliography

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(nlogn)$ sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 1–9. ACM, 1983.

[2] AMPL. AMPL - Streamlined Modeling for Real Optimization. http://ampl.com/.

[3] J. E. Atkins, E. G. Boman, and B. Hendrickson. A spectral algorithm for seriation and the consecutive ones problem. *SIAM Journal on Computing*, 28(1):297–310, 1998.

[4] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, pages 307–314. ACM, 1968.

[5] COR@L. Lehigh ISE COR@L Lab Wiki. http://coral.ie.lehigh.edu/wiki/doku.php/coral_lab.

[6] T. H. Cormen. In *Introduction to Algorithms*, chapter 27, pages 704–724. MIT press, 2009.

[7] M. Fiori, P. Sprechmann, J. Vogelstein, P. Musé, and G. Sapiro. Robust multimodal graph matching: Sparse coding meets graph matching. In *Advances in Neural Information Processing Systems*, pages 127–135, 2013.

[8] F. Fogel, R. Jenatton, F. Bach, and A. d'Aspremont. Convex relaxations for permutation problems. In *Advances in Neural Information Processing Systems*, pages 1016–1024, 2013.

[9] G. C. Garriga, E. Junttila, and H. Mannila. Banded structure in binary matrices. *Knowledge and Information Systems*, 28(1):197–226, 2011.

[10] A. George and A. Pothen. An analysis of spectral envelope reduction via quadratic assignment problems. *SIAM Journal on Matrix Analysis and Applications*, 18(3):706–732, 1997.

[11] M. X. Goemans. Smallest compact formulation for the permutahedron. *Mathematical Programming*, 153(1):5–11, 2015.

[12] F. R. Hodson. *The La Tène cemetery at Münsingen-Rain: catalogue and relative chronology*, volume 5. Stämpfli, 1968.

[13] IBM ILOG. High-performance mathematical programming solver for linear programming, mixed integer programming, and quadratic programming. https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

[14] D. G. Kendall. Abundance matrices and seriation in archaeology. *Probability Theory and Related Fields*, 17(2):104–112, 1971.

[15] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society*, pages 497–520, 1960.

[16] I. Liiv. Seriation and matrix reordering methods: an historical overview. *Statistical Analysis and Data Mining*, 3(2):70–91, 2010.

[17] C. H. Lim and S. J. Wright. Sorting network relaxations for vector permutation problems. *arXiv preprint arXiv:1407.6609*, 2014.

[18] R. Martí and G. Reinelt. *The Linear Ordering Problem: Exact and Heuristic Methods in Combinatorial Optimization*, volume 175. Springer Science & Business Media, 2011.

[19] J. Meidanis, O. Porto, and G. P. Telles. On the consecutive ones property. *Discrete Applied Mathematics*, 88(1-3):325–354, 1998.

[20] W. S. Robinson. A method for chronologically ordering archaeological deposits. *American Antiquity*, 16(4):293–301, 1951.

[21] Wikipedia. Bitonic sorter — Wikipedia, the Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Bitonic_sorter&oldid=748679110, 2016. [Online; accessed 30-April-2017].

[22] Wikipedia. Branch and bound — Wikipedia, the Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Branch_and_bound&oldid=764241953, 2017. [Online; accessed 5-May-2017].

[23] Wikipedia. Integer programming — Wikipedia, the Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Integer_programming&oldid=778379010, 2017. [Online; accessed 5-May-2017].

Appendix A

AMPL Model for input datasize of 8

```
set N:= 1..8;

set CN:= 1..24;

param M{i in N, j in N};

param Perm{i in N};

var X0{i in N} >= 1, <= 8;
var X1{i in N} >= 1, <= 8;
var X2{i in N} >= 1, <= 8;
var X3{i in N} >= 1, <= 8;
var X4{i in N} >= 1, <= 8;
var X5{i in N} >= 1, <= 8;
var X6{i in N} >= 1, <= 8;
var C{i in CN} binary;
```

```
minimize objfunc: sum{i in N, j in N} 2 * X0[i] * M[i,j] * X0[j];

subject to X0toX1_1: X1[1] <= X0[1];
subject to X0toX1_2: X1[1] <= X0[2];
subject to X0toX1_3: X1[1] >= X0[1] - 8*C[1];
subject to X0toX1_4: X1[1] >= X0[2] - 8 * (1 - C[1]);
subject to X0toX1_5: X1[2] = X0[1] + X0[2] - X1[1];
subject to X0toX1_6: X1[3] <= X0[3];
subject to X0toX1_7: X1[3] <= X0[4];
subject to X0toX1_8: X1[3] >= X0[3] - 8*C[2];
subject to X0toX1_9: X1[3] >= X0[4] - 8 * (1 - C[2]);
subject to X0toX1_10: X1[4] = X0[3] + X0[4] - X1[3];
subject to X0toX1_11: X1[5] <= X0[5];
subject to X0toX1_12: X1[5] <= X0[6];
subject to X0toX1_13: X1[5] >= X0[5] - 8*C[3];
subject to X0toX1_14: X1[5] >= X0[6] - 8 * (1 - C[3]);
subject to X0toX1_15: X1[6] = X0[5] + X0[6] - X1[5];
subject to X0toX1_16: X1[7] <= X0[7];
subject to X0toX1_17: X1[7] <= X0[8];
subject to X0toX1_18: X1[7] >= X0[7] - 8*C[4];
subject to X0toX1_19: X1[7] >= X0[8] - 8 * (1 - C[4]);
subject to X0toX1_20: X1[8] = X0[7] + X0[8] - X1[7];
subject to C1: sum{i in 1..4} C[i] <= 4;

subject to X1toX2_1: X2[1] <= X1[1];
subject to X1toX2_2: X2[1] <= X1[4];
subject to X1toX2_3: X2[1] >= X1[1] - 8*C[5];
subject to X1toX2_4: X2[1] >= X1[4] - 8 * (1 - C[5]);
subject to X1toX2_5: X2[4] = X1[1] + X1[4] - X2[1];
```

```
subject to X1toX2_6: X2[2] <= X1[2];
subject to X1toX2_7: X2[2] <= X1[3];
subject to X1toX2_8: X2[2] >= X1[2] - 8*C[6];
subject to X1toX2_9: X2[2] >= X1[3] - 8 * (1 - C[6]);
subject to X1toX2_10: X2[3] = X1[2] + X1[3] - X2[2];
subject to X1toX2_11: X2[5] <= X1[5];
subject to X1toX2_12: X2[5] <= X1[8];
subject to X1toX2_13: X2[5] >= X1[5] - 8*C[7];
subject to X1toX2_14: X2[5] >= X1[8] - 8 * (1 - C[7]);
subject to X1toX2_15: X2[8] = X1[5] + X1[8] - X2[5];
subject to X1toX2_16: X2[6] <= X1[6];
subject to X1toX2_17: X2[6] <= X1[7];
subject to X1toX2_18: X2[6] >= X1[6] - 8*C[8];
subject to X1toX2_19: X2[6] >= X1[7] - 8 * (1 - C[8]);
subject to X1toX2_20: X2[7] = X1[6] + X1[7] - X2[6];
subject to C2: sum{i in 5..8} C[i] <= 4;

subject to X2toX3_1: X3[1] <= X2[1];
subject to X2toX3_2: X3[1] <= X2[2];
subject to X2toX3_3: X3[1] >= X2[1] - 8*C[9];
subject to X2toX3_4: X3[1] >= X2[2] - 8 * (1 - C[9]);
subject to X2toX3_5: X3[2] = X2[1] + X2[2] - X3[1];
subject to X2toX3_6: X3[3] <= X2[3];
subject to X2toX3_7: X3[3] <= X2[4];
subject to X2toX3_8: X3[3] >= X2[3] - 8*C[10];
subject to X2toX3_9: X3[3] >= X2[4] - 8 * (1 - C[10]);
subject to X2toX3_10: X3[4] = X2[3] + X2[4] - X3[3];
subject to X2toX3_11: X3[5] <= X2[5];
subject to X2toX3_12: X3[5] <= X2[6];
```

```
subject to X2toX3_13: X3[5] >= X2[5] - 8*C[11];
subject to X2toX3_14: X3[5] >= X2[6] - 8 * (1 - C[11]);
subject to X2toX3_15: X3[6] = X2[5] + X2[6] - X3[5];
subject to X2toX3_16: X3[7] <= X2[7];
subject to X2toX3_17: X3[7] <= X2[8];
subject to X2toX3_18: X3[7] >= X2[7] - 8*C[12];
subject to X2toX3_19: X3[7] >= X2[8] - 8 * (1 - C[12]);
subject to X2toX3_20: X3[8] = X2[7] + X2[8] - X3[7];
subject to C3: sum{i in 9..12} C[i] <= 4;



subject to X3toX4_1: X4[1] <= X3[1];
subject to X3toX4_2: X4[1] <= X3[8];
subject to X3toX4_3: X4[1] >= X3[1] - 8*C[13];
subject to X3toX4_4: X4[1] >= X3[8] - 8 * (1 - C[13]);
subject to X3toX4_5: X4[8] = X3[1] + X3[8] - X4[1];
subject to X3toX4_6: X4[2] <= X3[2];
subject to X3toX4_7: X4[2] <= X3[7];
subject to X3toX4_8: X4[2] >= X3[2] - 8*C[14];
subject to X3toX4_9: X4[2] >= X3[7] - 8 * (1 - C[14]);
subject to X3toX4_10: X4[7] = X3[2] + X3[7] - X4[2];
subject to X3toX4_11: X4[3] <= X3[3];
subject to X3toX4_12: X4[3] <= X3[6];
subject to X3toX4_13: X4[3] >= X3[3] - 8*C[15];
subject to X3toX4_14: X4[3] >= X3[6] - 8 * (1 - C[15]);
subject to X3toX4_15: X4[6] = X3[3] + X3[6] - X4[3];
subject to X3toX4_16: X4[4] <= X3[4];
subject to X3toX4_17: X4[4] <= X3[5];
subject to X3toX4_18: X4[4] >= X3[4] - 8*C[16];
```

```
subject to X3toX4_19: X4[4] >= X3[5] - 8 * (1 - C[16]);
subject to X3toX4_20: X4[5] = X3[4] + X3[5] - X4[4];
subject to C4: sum{i in 13..16} C[i] <= 4;


subject to X4toX5_1: X5[1] <= X4[1];
subject to X4toX5_2: X5[1] <= X4[3];
subject to X4toX5_3: X5[1] >= X4[1] - 8*C[17];
subject to X4toX5_4: X5[1] >= X4[3] - 8 * (1 - C[17]);
subject to X4toX5_5: X5[3] = X4[1] + X4[3] - X5[1];
subject to X4toX5_6: X5[2] <= X4[2];
subject to X4toX5_7: X5[2] <= X4[4];
subject to X4toX5_8: X5[2] >= X4[2] - 8*C[18];
subject to X4toX5_9: X5[2] >= X4[4] - 8 * (1 - C[18]);
subject to X4toX5_10: X5[4] = X4[2] + X4[4] - X5[2];
subject to X4toX5_11: X5[5] <= X4[5];
subject to X4toX5_12: X5[5] <= X4[7];
subject to X4toX5_13: X5[5] >= X4[5] - 8*C[19];
subject to X4toX5_14: X5[5] >= X4[7] - 8 * (1 - C[19]);
subject to X4toX5_15: X5[7] = X4[5] + X4[7] - X5[5];
subject to X4toX5_16: X5[6] <= X4[6];
subject to X4toX5_17: X5[6] <= X4[8];
subject to X4toX5_18: X5[6] >= X4[6] - 8*C[20];
subject to X4toX5_19: X5[6] >= X4[8] - 8 * (1 - C[20]);
subject to X4toX5_20: X5[8] = X4[6] + X4[8] - X5[6];
subject to C5: sum{i in 17..20} C[i] <= 4;



subject to X5toX6_1: X6[1] <= X5[1];
subject to X5toX6_2: X6[1] <= X5[2];
```

```
subject to X5toX6_3: X6[1] >= X5[1] - 8*C[21];
subject to X5toX6_4: X6[1] >= X5[2] - 8 * (1 - C[21]);
subject to X5toX6_5: X6[2] = X5[1] + X5[2] - X6[1];
subject to X5toX6_6: X6[3] <= X5[3];
subject to X5toX6_7: X6[3] <= X5[4];
subject to X5toX6_8: X6[3] >= X5[3] - 8*C[22];
subject to X5toX6_9: X6[3] >= X5[4] - 8 * (1 - C[22]);
subject to X5toX6_10: X6[4] = X5[3] + X5[4] - X6[3];
subject to X5toX6_11: X6[5] <= X5[5];
subject to X5toX6_12: X6[5] <= X5[6];
subject to X5toX6_13: X6[5] >= X5[5] - 8*C[23];
subject to X5toX6_14: X6[5] >= X5[6] - 8 * (1 - C[23]);
subject to X5toX6_15: X6[6] = X5[5] + X5[6] - X6[5];
subject to X5toX6_16: X6[7] <= X5[7];
subject to X5toX6_17: X6[7] <= X5[8];
subject to X5toX6_18: X6[7] >= X5[7] - 8*C[24];
subject to X5toX6_19: X6[7] >= X5[8] - 8 * (1 - C[24]);
subject to X5toX6_20: X6[8] = X5[7] + X5[8] - X6[7];
subject to C6: sum{i in 21..24} C[i] <= 4;


subject to X6_final{i in N} : X6[i] = i;
```

Appendix B

Python code to build AMPL files

```
"""
    this code is to generate data and model file for AMPL
    of arbitrary length of input permutation
    Last edit: Feb 26, 2017
        Mar 26: This code is updated to modify the correlation variable C[...]
            so that for each depth the index of C does not started over from 1.
"""
import numpy as np
from math import log
import time

"""START BUILDING SORTING NETWORK"""
def merger(n,stage):
    #print 'this is the %d-th stage' %(stage)
    comparators=[]
    Groups = n/(2**stage)
    #print 'Groups =%d ' %(Groups)

    for group in range(0,Groups):
```

```
        listToClean = range(group*(n/Groups),group*n/Groups+n/Groups)
        # the first half-cleaner of each group is revised
        for j in range(0,len(listToClean)/2):
            comparators.append([listToClean[j],listToClean[len(listToClean)-j-1]])

    for depth in range(1,stage):
        for group in range(0,Groups):
            #print 'this is the %d-th group' %(group)
            listToClean = range(group*(n/Groups),group*n/Groups+n/Groups)

            length = len(listToClean)
            #print listToClean

            Groups2 = 2**depth
            for group2 in range(0,Groups2):
                for i in range(group2*length/Groups2,group2*length/Groups2+length/Groups2/2):
                        #print [listToClean[i],listToClean[i+length/Groups2/2]]
                    comparators.append([listToClean[i],listToClean[i+length/Groups2/2]])
    #print comparators
    #print '----------------'
    return comparators

def builtNetwork(n,depth,len_perm):
    comparators=[]
    sortingNetwork = []

    for i in range(0,n-1,2):
        comparators.append([i,i+1])
```

```
    # the i_th stage(from the 2nd stage)
    for stage in range(2,int(log(n,2))+1):
        comparators += merger(n,stage)

    #group comparators by depth
    for i in range(0,depth):
        sortingNetwork.append(comparators[(i*n/2):(i*n/2+n/2)])
    sortingNetwork
    # remove comparators whose index is larger than length of permutation
    numOfComparators = n/2 * depth
    for i in range(0,depth):
        for j in range(len(sortingNetwork[i])-1,-1,-1):
            if max(sortingNetwork[i][j])>(len_perm-1):
                sortingNetwork[i].remove(sortingNetwork[i][j])
                numOfComparators -= 1
    return sortingNetwork, numOfComparators
"""END BUILDING SORTING NETWORK"""


"""GENERATE DATA FILE FOR AMPL"""
def generateDataFile(N,M):
    '''
    #This part is to generate matrix according
    #to the input length of the matrix
    #generate Matrix M
    np.random.seed(7)
    A = np.random.rand(N, 2*N)
    M = np.dot(A, A.transpose())
    M = M/np.max(M)
```

```
D = np.diag(np.diag(M**(-0.5)))
M = np.dot(np.dot(D,M),D)

#write matrix M to file for MATLAB
fw = open('matrixA.dat','w')
for raw in range(0,N):
    np.savetxt(fw,[M[raw]],fmt='%-8f')
fw.close()
'''
#write matrix and permutation into data file for AMPL
perm = []
perm += [i for i in range(1,N+1)]

M = np.insert(M, 0, perm, axis = 1)

f = open('sortingNetwork.dat','w')
list = ' '
for i in range(1,N+1):
    list = list + str(i) +' '
f.write('param Perm: %s :=\n' % list)
f.write('1 %s\n;\n\n' % list)
f.write('param M: '+ ' '.join(str(x) for x in perm) + ':=\n')
for raw in range(0,len(M)):
    np.savetxt(f,[M[raw]],fmt='%-8f')
    if raw == N-1:
        f.write(';\n\n')
f.close()

return
```

```
"""GENERATE MODEL FILE FOR AMPL"""
def generateModFile(len_perm,N,depth,sortingNetwork,numOfComparators):
    CN = numOfComparators
    f = open('sortingNetwork.mod','w')
    f.write('set N:= 1..%d;\n\n' % len_perm)
    f.write('set CN:= 1..%d;\n\n' % CN)
    f.write('param M{i in N, j in N};\n\n')
    f.write('param Perm{i in N};\n\n')
    # generate variable list
    for i in range(0,depth+1):
        f.write('var X%d{i in N} integer >= 1, <= %d;\n' % (i, len_perm))
    f.write('var C{i in CN} binary;\n')

    # generate objective function
    f.write('\nminimize objfunc: sum{i in N, j in N} X0[i] * M[i,j] * X0[j];\n\n')

    # generate constraints
    numOfC = 0
    for i in range(0,depth):
        num = len(sortingNetwork[i])
        marker = list(range(1,len_perm+1)) # to mark elements not compared in this depth,
        to pass the value of elements in marker

        for j in range(0,num):

            f.write('subject to X%dtoX%d_%d: X%d[%d] <= X%d[%d];\n' \
                    %(i, i+1, 5*j+1, i+1, sortingNetwork[i][j][0]+1, i,
```

```
                    sortingNetwork[i][j][0]+1))

            f.write('subject to X%dtoX%d_%d: X%d[%d] <= X%d[%d];\n' \
                    %(i, i+1, 5*j+2, i+1, sortingNetwork[i][j][0]+1, i,
                    sortingNetwork[i][j][1]+1))

            f.write('subject to X%dtoX%d_%d: X%d[%d] >= X%d[%d] - %d*C[%d];\n' \
                    %(i, i+1, 5*j+3, i+1, sortingNetwork[i][j][0]+1, i,
                    sortingNetwork[i][j][0]+1, N, numOfC+j+1))
#               numOfC += 1

            f.write('subject to X%dtoX%d_%d: X%d[%d] >= X%d[%d] - %d * (1 - C[%d]);\n' \
                    %(i, i+1, 5*j+4, i+1, sortingNetwork[i][j][0]+1, i,
                    sortingNetwork[i][j][1]+1, N, numOfC+j+1))
#               numOfC += 1

            f.write('subject to X%dtoX%d_%d: X%d[%d] = X%d[%d] + X%d[%d] - X%d[%d];\n' \
                    %(i, i+1, 5*j+5, i+1, sortingNetwork[i][j][1]+1, i,
                    sortingNetwork[i][j][0]+1,\
                    i, sortingNetwork[i][j][1]+1, i+1,
                    sortingNetwork[i][j][0]+1) )
            marker.remove(sortingNetwork[i][j][0]+1);
            marker.remove(sortingNetwork[i][j][1]+1);
        num_eq = 5*j+5
        if len_perm != N:
            for e in marker:
                num_eq += 1
                f.write('subject to X%dtoX%d_%d: X%d[%d] = X%d[%d];\n' \
                        %(i, i+1, num_eq, i+1, e,i, e) )
```

```
        else:
            pass #the error before was caused by using continue here


        f.write('subject to C%d: sum{i in %d..%d} C[i] <= %d;\n\n' \
                %(i+1, numOfC+1, numOfC+num, num))
        numOfC += num
        f.write('\n')
    f.write('subject to X%d_final{i in N} : X%d[i] = i;\n\n' \
            %(depth, depth) )



    f.close()
    return


"""GENERATE RUN FILE FOR AMPL"""
def generateRunFile():
    f = open('sortingNetwork.run','w')
    f.write('reset;\n\n')
    f.write('option solver \'/usr/local/cplex/bin/x86-64_linux/cplexamp\';\n\n')
    f.write('option cplex_options \'mipdisplay=2 mipgap=0.01\';\n\n')
    f.write('model sortingNetwork.mod;\n\n')
    f.write('data sortingNetwork.dat;\n\n')
    f.write('solve;\n\n')
    f.write('display _solve_system_time;\n\n')
    f.write('X0;\n\n')
    f.close()
    return
```

```
"""MAIN FUNCTION"""
def main():
#     len_perm = input('The length of permutation is: ')
    M = np.loadtxt('8_1_lap.dat')

    len_perm = np.shape(M)[0]
    print 'The lengh of permutation is: ',len_perm
    # N is the size of input for sorting network
    if log(len_perm,2)-round(log(len_perm,2))==0:
        N = len_perm
    else:
        N = 2**(int(log(len_perm,2))+1)
    lgN = log(N,2)

    depth = int(0.5 * (1+lgN) * lgN)
    sortingNetwork, numOfComparators = builtNetwork(N,depth,len_perm)
#     print sortingNetwork
    start = time.time()
    generateDataFile(len_perm,M)
    generateModFile(len_perm,N,depth,sortingNetwork,numOfComparators)
    generateRunFile()
    end = time.time()
    print 'Time to generate AMPL files is:', end - start
#     print sortingNetwork


if __name__ == '__main__':
    main()
```

Biography

Yutong Chang was born November 12, 1993 in Rizhao, Shandong, China. She attended No.1 High School in Rizhao. During her undergraduate program at Northwestern Polytechnical University in China, Yutong was majored in Engineering Mechanics, and in June of 2015 received her degree in Bachelor of Science in Engineering Mechanics. During her undergraduate, she did several research projects as team leader, funded by The National College Students Innovative Training Program and The National College Students Innovative Training Program. She then attended the graduate program in Department of Industrial Systems and Engineering, and will receive her Master of Science in Management Science and Engineering in May of 2017.